ENSURING HIGH PERFORMANCE IN VALIDATING XML PARSER

Donglei Cao, Shuang Yu, Beijie Dai and Beihong Jin
Technology Center of Software Engineering, Institute of Software
Chinese Academy of Sciences, Beijing 100080, China

Keywords: XML Parser, Validation, Performance Tuning.

Abstract: An XML parser is the fundamental software for analyzing and processing XML documents. This paper presents the optimized validation algorithms in OnceXMLParser, a full-validating XML Parser. OnceXMLParser adopts a lightweight architecture and implements several efficient algorithms for validating. Since the element validating is a great challenge to the performance of a validating XML parser, this paper focused on two key algorithms to resolve it. The first one involves in an optimized automaton used to build these element validating rules efficiently. The second one is a statistical predictive algorithm to reduce the name string recognizing process. For a valid document, this algorithm could make precise prediction when the child elements are sequentially defined, and could fulfil the least cost prediction according to the cost function when the child elements are optionally defined. Performance testing shows OnceXMLParser after performance tuning has outstanding parsing efficiency.

1 INTRODUCTION

The Extensible Markup Language (XML) (W3C, 1998) has been widely used in electronic businesses, web services and enterprise data exchanges and integrations; it actually becomes a standard for data representation and exchange over network. In addition, as a meta-language, XML can be used to define a wide range of markup languages, such as Web Service Definition Language, Resource Description Framework and Mathematical Markup Language. However, text-based format of XML sometimes leads to a large document. Moreover, XML-based data exchange happens much frequently in some scenarios, for example, in Web service applications. Thus, XML parsing in those applications will become a system bottleneck which demands to improve the performance of XML parsers.

This paper gives the optimization techniques of validity checking in OnceXMLParser which is a high-performance full-validating XML Parser supporting Simple API for XML (SAX), Document Object Model (DOM) and Streaming API for XML (StAX). The paper is organized as follows. Section 2 introduces some popular XML parsers; section 3 gives the system architecture of OnceXMLParser; section 4 presents key validity checking algorithms; section 5 shows our test results, mainly in StAX, and at last is the summary.

2 RELATED WORK

There are many popular XML parsers for Java, such as Xerces (Apache, 2004), Crimson (Apache, 2001), Piccolo (Oren, 2002), etc. Among these parsers, Xerces is the most popular validating XML parser supporting DOM and SAX, and it is also the default parser since JDK 5.0. Crimson supports XML1.0, SAX2, DOM Core 2 and JAXP1.1. Piccolo is a validating SAX parser which is generated by parser generator tools JFlex and BYACC/J. However, Crimson is not perfect in parsing; for example, it cannot recognize byte order mark in UTF-8 coding, and invalid character references in attribute values. Similarly, Piccolo also has defects in parsing and validating, for instance, it cannot resolve relative paths in entity references correctly, and does not check the validity constraint (VC): Proper Conditional Section/PE Nesting. In brief, Xerces is excellent in SAX and DOM although it enters loop after reading 10M-size comment or PI.

Cao D., Yu S., Dai B. and Jin B. (2007).
ENSURING HIGH PERFORMANCE IN VALIDATING XML PARSER.
In *Proceedings of the Third International Conference on Web Information Systems and Technologies - Internet Technology*, pages 210-215
DOI: 10.5220/0001276102100215

Recently, StAX, a promising pull API for XML processing, is implemented by some XML parsers, for example BEA StAX RI (BEA, 2003), Sun Java Streaming XML Parser (Sun, 2005), Oracle StAX Pull Parser (Oracle, 2003) and Woodstox (Codehaus, 2006). But all of them have some fatal defects. For example, BEA StAX RI cannot recognize invalid characters in character data part of the document, cannot parse entity references in the default attribute values and cannot get the correct character text when it reports a characters event; Sun Java Streaming XML Parser cannot correctly deal with the external parameter entity references in Document Type Definition (DTD), cannot recognize legal characters ranged from #x10000 to #x10FFFF and surrogate pairs, and it doesn't read attribute-list declarations in the external subset; Oracle StAX Pull Parser is based on the SAX Parser of Oracle XDK which leads to its inefficient; Woodstox cannot fully support UTF-8, especially the surrogate pairs, and it cannot recognize some invalid names. As the only validating parser among those StAX parsers, Woodstox doesn't report errors when it meets invalid XML documents; instead it reports exceptions and stops parsing.

Besides these common XML parsers, some researches also focus on automatically generating XML parsers from an XML Schema. XML Screamer (Kostoulas et al, 2006) is such an experimental system. Its parsing is integrated with Schema-based validation and deserialization to achieve high performance. But these automatically generated parsers are not common XML parsers. Each generated parser only fits those Schema-based valid documents under a particular XML Schema.

OnceXMLParser is a common XML parser and fully implements XML1.0/1.1 and Namespaces in XML (W3C, 1999). It passes all the API tests for StAX (Tatu, 2004), DOM (W3C, 2004) and SAX (David, 2001) and all the XML conformance tests (W3C, 2003), it also shows outstanding parsing performance after adopting some efficient performance tuning algorithms.

3 SYSTEM ARCHITECTURE

OnceXMLParser adopts a lightweight architecture and consists of the following components shown in Figure 1.

Like common lexical analysis used for traditional programming language, Scanner is implemented as a Deterministic Finite Automaton (DFA) to recognize terminals from the input streams. Some well-formedness constraints (WFCs), such as checking valid name characters and valid attribute values, are checked while recognizing those tokens. But before

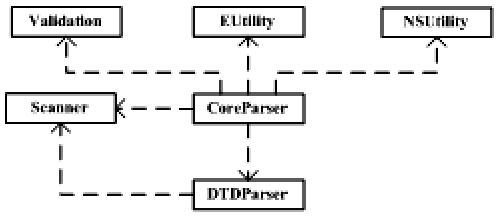


Figure 1: Architechture of OnceXMLParser.

regular lexical analysis, Scanner has to decode characters denoted by encoding signature in the scanning buffer and then put the decoded characters into a pre-allocated character buffer. Considering that the smaller size of character buffer needs more filling buffer operations and more saving actions for unparsed tokens at the end of character buffer, but the larger size of character buffer increases costs for normalization operations, we adjust the proper size of character buffer in order to improve performance.

CoreParser is the core syntax processor, which uses tokens got from Scanner and returns interested information to applications through some standard APIs. In addition, CoreParser checks most WFCs. For example, in order to conform to WFC: Element Type Match (W3C, 1998), CoreParser must check whether these element names in start tags and corresponding end tags are equal. CoreParser resolves markups and characters according to their statistical frequencies of occurrences, first for the highest frequency.

DTDParser is a syntax analyzer for processing declarations in DTD. It is also responsible for building VC rules. In order to make the building process efficiently, especially for those VC rules about elements' relations, DTDParser needs efficient algorithms which will be discussed in the next section. In addition, in order to facilitate the building and checking processes of VC rules, efficient data structures, such as light-weighed hash map, are implemented.

Validation module checks VC rules collected by DTDParser. The design of validation module is crucial for performance, because these checking are always burdensome tasks, for example, VC rules of elements usually concern with the relationships among elements and the content's format of a particular element; rules of attributes always concern with the type and the default value of this attribute. Moreover the efficient checking of VC: Element Valid (W3C, 1998) is the largest challenge as we will discuss soon, so we present some key algorithms to solve it in the next section.

NSUtility resolves namespace declarations and manages namespace scopes and mappings. It also takes charge of namespace constraints (NSC) checking. For example, to conform to NSC: Prefix Declared (W3C, 1999), NSUtility must check whether the prefix *xml* is bound to *http://www.w3.org/XML/1998/namespace* or not. EUtility manages entities referenced in the document. It also has to check some WFCs, such as WFC: No Recursion (W3C, 1998).

4 ALGORITHMS FOR VALIDATING

Checking VC: Element Valid (W3C, 1998) means examining whether elements occurred in the document can match the declarations in DTD. Moreover, validity checking contains many reiterating processing, such as comparing name strings. The following algorithms aim to improve the performance of validity checking.

4.1 Optimized Automaton for Element Valid

According to (W3C, 1998), the element content, which defines the relationship of elements, is defined by a regular language with ε-operations such as * and *?*, so it is natural to build a Nondeterministic Finite Automaton (NFA) to represent and check these definitions. Of course, we could build a DFA as Woodstox did. But DFA usually has more states than NFA. Moreover, to build a DFA must translate these definitions with ε-operations to the equivalent forms containing no ε-operations. This will cause additional costs. At last, we should notice that the relations of elements appeared in the document are usually quite simple, though their definitions would be rather complex. Each automaton, no matter DFA or NFA, takes charge of checking all the child elements of one element type declaration. Each automaton starts at its initial state, and transforms the state according to the state transition function. For a valid document, the automaton stops when the last child defined in the corresponding declaration is matched. In fact, DFA and NFA will traversal element content from the initial state to the final state. So an NFA with less building costs usually performs better than a DFA, while the checking costs of the two are approximately equal due to the relatively simple structure of an XML document.

In order to show the performance between OnceXMLParser with built-in NFA and Woodstox integrated with DFA, we choose dozens of XML documents where DTD parts occupy most portions. These DTDs only have element type declarations. Each declaration contains ten child elements, and 16% of these child elements are defined to be optional. We let the two parsers parse those documents for 1000 times, and record the time (for building NFA/DFA) they used respectively. We tabulate these results in Table 1. The values of the Count column denote the number of element type declarations in DTD.

Table 1: Tests of NFA and DFA.

Count	NFA (OnceXMLParser)	DFA (Woodstox)
1000	65188 ms	81141 ms
2000	124765 ms	163781 ms
5000	328110 ms	378297 ms
7000	373735 ms	578438 ms
10000	579203 ms	750407 ms

These data show that NFA can perform better than DFA in those situations for about 32.7% on average which supports our NFA solution in OnceXMLParser.

4.2 Statistical Predictive Algorithm

*Name*s, which are defined by P[5]: Name Product of (W3C, 1998), occur much frequently in a normal XML document. Moreover, name recognizing is an exhaustive process, because parser must check all the characters appeared in the name whether they are in valid name characters set defined by (W3C, 1998). Notice that in valid XML documents, all the element names have firstly appeared in the DTD, and they will then appear elsewhere in the document many times. So a reasonable idea is that we only check these names in DTD and cache all the checked names in a buffer. Once a name appeared in the document, we would look up in the buffer to find the corresponding name. If we could find out one then we can omit the valid name checking, otherwise the document is invalid. This approach will introduce new costs in looking each name up in the buffer which should be controlled in a reasonable range. So we try to reduce the name recognizing by predicting the element name which is most likely to appear next according to current element name and some corresponding element type declarations. If we can provide relatively precise prediction, we need not to look up through the buffer or we could only need to look up in a small subset of the buffer.

Here is an example of element type declaration: *<!ELEMENT parentEle (seq1,seq2,(opt3|option))>*

This declaration means the parent element *parentEle* has three sequential children; they are *seq1*, *seq2* and one of the *opt3* and *option* in order. It is easy to know that after the element *parentEle*, there must be an element *seq1*. Following *seq1* there must be *seq2*, because of the sequential definition. After *seq2* we have two choices, *opt3* and *option*, which are defined to appear optionally. We must get the best prediction according to our cost function (1), which takes the lengths of names as its parameters.

$$C(name) = \frac{1}{n} \times C_{success} + \sum \frac{1}{n} \times C_{failure} \tag{1}$$

Where n is the number of options in the element type declaration which is recorded by DTDParser — in our example n equals 2. $C_{success}$ and $C_{failure}$ stand for the costs of a successful prediction and a failed prediction respectively. So *C(name)* is the expectation of the cost of the prediction process. Then we can evaluate each choice by the cost function to decide the one cost least.

$C_{success}$ is only concerned with the length of the predicted name. Considering Scanner will compare current characters in the character buffer with those in the given predicted name. $C_{success}$ denotes the cost of the comparing. In the case of a successful prediction, Scanner compares each character in character buffer with corresponding characters in the predicted name, liking string matching, till the end of the predicted name. So the number of characters read by Scanner is the length of the given name exactly, therefore $C_{success}$ is only concerned with the length of the given name without respect to what the name is, namely $C_{success} = C_{success}(name_{predict}.length) = C_{success}(name_{real}.length)$, where $name_{predict}$ means the predicted name and $name_{real}$ means the real name stored in the character buffer. On the other hand, $C_{failure}$ does not only concern with the predicted name only but is also concerns with the real name in the character buffer. In the case of failed prediction, there must be a character in the character buffer, which does not match the corresponding character in the predicted name. And this mismatch may happen at any position during the comparing, so the cost of this process also concerns with the real name, namely $C_{failure} = C_{failure}(name_{real}.length , name_{predict}.length)$. Notice that the real name must be one of the options in the element type declaration, since we assumed that the XML document is valid. To evaluate the values of $C_{success}$ and $C_{failure}$, we compute the comparing costs of vast different name strings with various lengths and get the statistical values. We can also establish the statistical values of the cost of processing a name in the normal way (without prediction), that means the cost of reading several characters from character buffer and looking up them in the valid name character set. Remember that, with prediction, Scanner compares each character in character buffer with just one corresponding character in the predicted name. But without prediction, Scanner compares each character with delimiters of the valid name character set. Obviously, for each character, the later operation costs more than the former operation. And it is easy to know that the cost is only concerned with the length of the name string analogous to $C_{success}$. We denote this as $C_{nonpredict} = C_{nonpredict}(name_{real}.length)$.

We choose words in Oxford Dictionary as our name strings. So we got 34,840 names with lengths ranged between 1 and 21, we could put names longer than 21 characters to the group of length 21.

Let G_i be the set of name strings whose length is i, i<21, G_{21} be the set of name strings whose length is not less than 21. Let *CountTime(process)* be the function to count the time used by the *process*, and *Sizeof(set)* be the function to get the number of elements of the *set*. Let getName($name_{real}$) be the process of recognizing a name string $name_{real}$ by the normal way (without prediction), skip($name_{predict}$) be the process of trying to match a predicted name $name_{predict}$, this process may be failed when the predicted name mismatch the real name in the document. Then we can evaluate $C_{nonpredict}$(i), $C_{success}$ (i) and $C_{failure}$(i,j) as:

$$C_{nonpredict}(i) = \frac{\sum_{name_{real} \in G_i} CountTime(getName(name_{real}))}{Sizeof(G_i)}$$

$$C_{success}(i) = \frac{\sum_{name_{real} \in G_i} CountTime(skip(name_{real}))}{Sizeof(G_i)} \tag{2}$$

$$C_{failure}(i, j) = \frac{\sum_{name_{predict} \in G_j, \forall name_{real} \in G_i} CountTime(skip(name_{predict}) + getName(name_{real}))}{Sizeof(G_j)}$$

Notice that, when computing $C_{failure}$(i,j), we randomly select a name string $name_{real}$ from G_i. If i equals to j, we guarantee $name_{real}$ is different from $name_{predict}$. We implement this algorithm in Java, and our experiment environment is a Pentium4 2.4GHz PC with 512M DDR, Windows XP, and JDK 1.4.2. We got the following statistical value tables, the unit of cost is10^{-4} millisecond. In Table 2, len denotes $name_{real}.length$, and in Table 3 value of the ith row and the jth column denotes the value of $C_{failure}$(i,j).

Table 2: Parts of the statistical values of $C_{success}$ and $C_{nonpredict}$.

L en	$C_{success}$	$C_{nonpredict}$	len	$C_{success}$	$C_{nonpredict}$
1	0.533	2.284	12	1.486	3.515
3	0.692	2.378	14	1.666	3.786
4	0.877	2.609	15	1.720	4.073
5	0.948	2.692	16	1.802	4.295
6	1.023	2.766	17	1.886	4.447
8	1.180	3.121	19	2.003	4.790
10	1.334	3.323	21	2.171	5.963

After the computation of *C(name)* comes into available, we can compute *C(opt3)* and *C(option)* in the former example respectively, and choose the one costs least as the predicted name. In our example,

$$C(opt3) = \frac{1}{2} \times C_{success}(opt3.length) + \frac{1}{2} \times C_{failure}(option.length, opt3.length) = \frac{1}{2}\times(C_{success}(4)+C_{failure}(6,4))=2.132$$

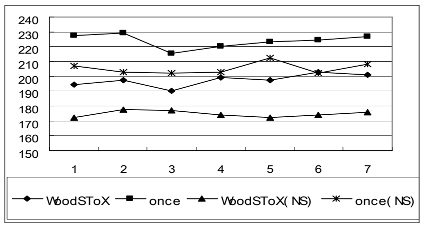
$$C(option) = \frac{1}{2} \times C_{cuccess}(option.length) + \frac{1}{2}\times C_{failure}(opt3.length, option.length) = \frac{1}{2}\times(C_{cuccess}(6)+C_{failure}(4,6))=2.0835$$

While $C_{nonpredict}(opt3.length) = C_{nonpredict}(4) = 2.609 > C(opt3)$ and $C_{nonpredict}(option.length) = C_{nonpredict}(6) = 2.766 > C(option)$. So we decide *option* as the predicted name, as it has the least cost according to our cost function (1).

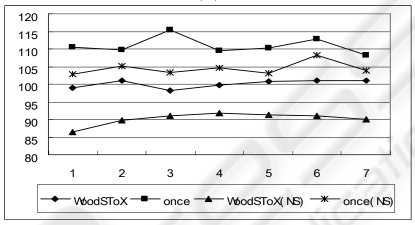
But if the one costing least is still larger than the corresponding $C_{nonpredict}$, the predicted name should be null, which means no prediction is the best way.

5 PERFORMANCE TESTING

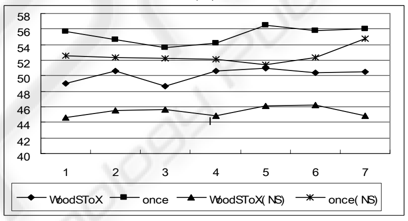
We use Sun XMLMark (Sun, 2004) as benchmark to compare the performance of OnceXMLParser with other XML parsers. We modified it to make StAX into use. To make the results stable, we repeat each test for seven times and get the average results. Our experiment environment is a Pentium4 2.8GHz PC (with Hyper Thread) and 1G DDR (Duel Channel), Window XP (SP2) and JDK 1.4.2_03. Details of Test1—Test3 can be found in (Sun, 2004).



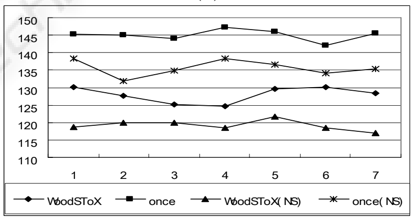
(a)



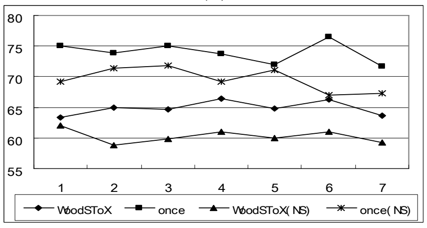
(b)



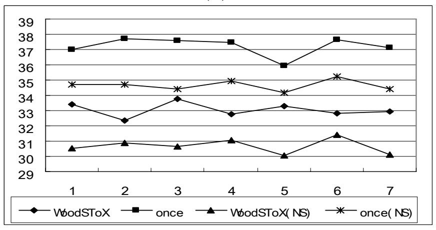
(c)



(d)



(e)



(f)

Figure 2: Results of XMLMark.

There are 24 groups of results, including 6 groups for SAX, 6 for StAX and 12 for DOM. Due to the limitation of page number; we only list 6 of them concerned with StAX here.

Testing results are shown in Figure 2. The horizon axis means the number of tests and the vertical axis means the number of finished transactions per second defined by (Sun, 2004).

In Figure 2, (a)—(c) mean the results without VC checking, while (d)—(f) mean the results with VC checking. In the figure, (NS) denotes the parsers support namespace.

For StAX, results show that OnceXMLParser with VC checking performs 13.32% and 14.10% better than Woodstox without and with namespace supporting respectively. And OnceXMLParser without VC checking performs 11.44% and 16.30% better than Woodstox without and with namespace supporting respectively. For SAX, results show that OnceXMLParser with VC checking performs 19.08% and 34.56% better than Xerces without and with namespace supporting respectively. And OnceXMLParser without VC checking performs 18.62% and 31.47% better than Xerces without and with namespace supporting respectively. For DOM, results show that OnceXMLParser with VC checking performs 38.04% and 31.41% better than Xerces without and with namespace supporting respectively. And OnceXMLParser without VC checking performs 77.44% and 57.08% better than Xerces without and with namespace supporting respectively.

6 CONCLUSION

XML parser is an infrastructure for XML processing. This paper studies some kinds of XML parser implements DOM, SAX and StAX, and provides a lightweight implementation—OnceXMLParser. Through implementing some key algorithms and some efficient optimizing techniques, OnceXMLParser gains better performance as we expected.

The future work includes supporting the XML Schema.

REFERENCES

W3C, 1998. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/1998/REC-xml-19980210*

W3C, 2004. DOM Conformance Test Suites, *http://www.w3.org/DOM/Test/*

W3C, 1999. Namespaces in XML. *http://www.w3.org/TR/1999/REC-xml-names-19990114*

W3C, 2003. Extensible Markup Language (XML) Conformance Test Suites 20031210. *http://www.w3.org/XML/Test/*

Tatu, S., 2004. StaxTest. *http://www.cowtowncoder.com/proj/staxtest*

Sun Microsystems, 2004. XML Test v1.1. *http://java.sun.com/performance/reference/codesamples*

David, B., 2001. SAX2Unit. *http://sourceforge.net/project/showfiles.php?group_id=8114&package_id=32032*

BEA, 2003. BEA RI. *http://ftpna2.bea.com/pub/downloads/jsr173.jar*

Sun Microsystems, 2005. Sun Java streaming XML parser. *https://sjsxp.dev.java.net/files/documents/3071/12956/sjsxp_20050505.class*

Codehaus, 2006. Woodstox.*http://woodstox.codehaus.org/*

Oracle, 2003. Oracle StAX Pull Parser, *http://www.oracle.com/technology/tech/xml/xdk/staxpreview.html*

Apache, 2004. Xerces2. *http://xml.apache.org/xerces2-j/*

Apache, 2001. Crimson. *http://xml.apache.org/crimson/*

Oren, Y., 2002. Piccolo. *http://piccolo.sourceforge.net/*

Kostoulas, G. M., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., Mercaldi, M., 2006. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *Proceedings of the 15th international conference on World Wide Web WWW '06*. ACM Press.

Table 3: Parts of the statistical values of $C_{failure}$(i,j).

	j=4	j=5	j=6	j=7	j=8	j=9	j=10	j=11	j=12	j=13	j=14
i=4	3.147	3.120	3.144	3.191	3.232	3.556	3.186	3.090	3.110	3.072	3.073
i=6	3.387	3.272	3.301	3.354	3.339	3.554	3.341	3.274	3.268	3.232	3.195
i=8	3.522	3.368	3.586	3.600	3.620	3.707	3.599	3.469	3.374	3.533	3.527
i=12	4.079	4.011	4.094	4.187	4.158	4.131	4.146	4.050	4.012	3.991	3.948